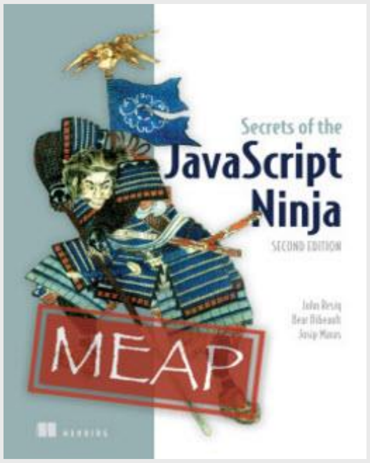


JavaScript: A look at Sets.

By John Resig, Bear Bibeault, and Josip Maras

In this article, excerpted from Secrets of the JavaScript Ninja, we take a look at a newcomer to JavaScript: Sets, collections of unique items.

In a large number of real-world problems, we often have to deal with collections of *distinct* items (meaning that each item cannot appear more than once), the so called *Sets*. Up to ES6, this was something that you had to implement yourself, by mimicking them with standard objects. For a very crude example, see the following listing.

Listing 1 Mimicking sets with objects

```
function Set(){
  this.data = {};                             //#A
  this.length = 0;                            //#A
}

Set.prototype.has = function(item){
  return typeof this.data[item] !== "undefined";
}

Set.prototype.add = function(item){
  if(!this.has(item)){                             //#B
    this.data[item] = true;                        //#B
    this.length++;                                 //#B
  }                                                //#B
};

Set.prototype.remove = function(item){
  if(this.has(item)){
    delete this.data[item];
    this.length--;
  }
}

var ninjas = new Set();
ninjas.add("Hattori");                                       //#C
ninjas.add("Hattori");                                    //#C

assert(ninjas.has("Hattori") && ninjas.length == 1,       //#D
       "Our set contains only one Hattori");              //#D

ninjas.remove("Hattori");
assert(!ninjas.has("Hattori") && ninjas.length == 0,
       "Our set is now empty");
```

#A Use an object to store items
#B Add an item only if it isn't already contained within the set
#C Try to add Hattori twice
#D Check that Hattori is added only once

Listing 1 shows a very simple example of how sets can be mimicked with objects. We use a data storage object `data`, for keeping track of our set items, and we've exposed three methods: the `has` method, which checks whether an item is already contained within our set; the `add` method, which adds an item, only if the same item is not already contained, and the `remove` method, which removes an already existing item from the set.

However, this is a poor doppelganger, as with maps, you cannot really store objects, only strings and numbers, and there's always the risk of accessing prototype objects. For these reasons, the ECMAScript comity has decided to introduce a completely new type of a collection *Sets*.

NOTE Sets are a part of the ES6 standard. For current browser compatibility, see: https://kangax.github.io/compat-table/es6/#test-Set.

Creating our first Set

The cornerstone of creating sets is the newly introduced constructor function, conveniently named `Set`. Let's see an example.

Listing 2 Creating sets

```
var ninjas = new Set(["Kuma", "Hattori", "Yagyu", "Hattori"]);         //#A

assert(ninjas.has("Hattori"), "Hattori is in our set");                //#B
assert(ninjas.size == 3, "There are only three ninjas in our set!");   //#B

assert(!ninjas.has("Yoshi"), "Yoshi is not in, yet..");               //#C
ninjas.add("Yoshi");                                                   //#C
assert(ninjas.has("Yoshi"), "Yoshi is added");                         //#C
assert(ninjas.size == 4, "There are four ninjas in our set!");          //#C

assert(ninjas.has("Kuma"), "Kuma is already added");                    //#D
ninjas.add("Kuma");                                                     //#D
assert(ninjas.size == 4, "Adding Kuma again has no effect");            //#D

for(var ninja of ninjas) {                                              //#E
  assert(ninja, ninja);                                                 //#E
}                                                                       //#E
```

#A The `Set` constructor can take in an array of items with which the set will be initialized
#B Any duplicate items are simply discarded
#C We can add new items, that aren't already contained within the set
#D Adding existing items will have no effect
#E We can iterate through sets with the `for…of` loop

In listing 2, we use the built-in `Set` constructor to create a new `ninjas` set that will contain distinct ninjas. If we don't pass in any arguments, an empty set will be created. We can also pass in an array, as we did in this example, which will prefill the set for us.

```
new Set(["Kuma", "Hattori", "Yagyu", "Hattori"]);
```

As we already mentioned, sets are collections of unique items, and their primary purpose is to stop us from storing multiple occurrences of the same object. In our case, this means that `"Hattori"`, which we tried to add twice, will only be added only once.

Every set has a number of methods accessible from it. For example, the `has` method checks whether an item is contained in the set

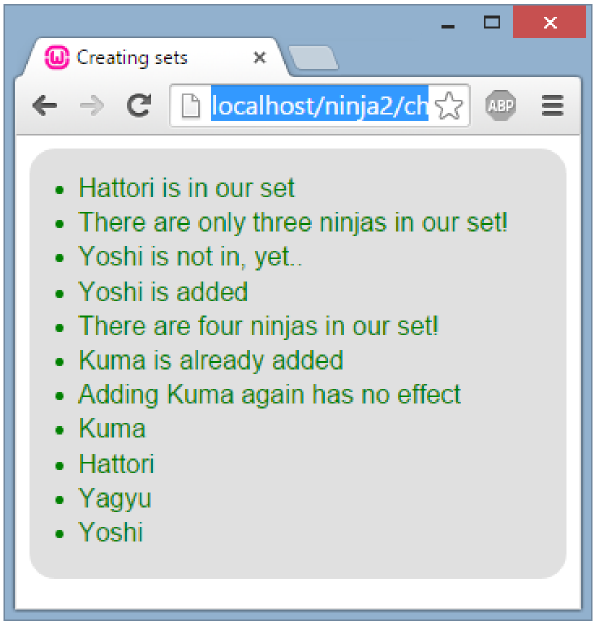
```
ninjas.has("Hattori")
```

and the `add` method is used to add unique items into the set:

```
ninjas.add("Yoshi");
```

If we are curious about how many items are there in a set, we can always use the `size` property.

Similar to maps and arrays, since sets are also collections, so there's nothing stopping us from iterating over them with a `for…of` loop. As you can see in the following figure, the items are always iterated over in the order in which they were inserted.



```
• Hattori is in our set
• There are only three ninjas in our set!
• Yoshi is not in, yet..
• Yoshi is added
• There are four ninjas in our set!
• Kuma is already added
• Adding Kuma again has no effect
• Kuma
• Hattori
• Yagyu
• Yoshi
```

Figure 1 Running the code from listing 2 produces the following output. The items in a set are always iterated over in the order in which they were inserted.

Now that we've gone through the basics of sets, let's visit some common operations on sets: unions, intersections, and differences.

Union of sets

The first operation that we're going to study is *union*. Simply put, a union of two sets `A` and `B`, creates a new set which contains all elements from both `A` and `B`. Naturally each item cannot occur more than once in the new set.

Listing 3 Using sets to perform a union of collections

```
var ninjas = ["Kuma", "Hattori", "Yagyu"];                  //#A
var samurai = ["Hattori", "Oda", "Tomoe"];                  //#A

var warriors = new Set([...ninjas, ...samurai]);            //#B

assert(warriors.has("Kuma"), "Kuma is here");                      //#C
assert(warriors.has("Hattori"), "And Hattori");                    //#C
assert(warriors.has("Yagyu"), "And Yagyu");                        //#C
assert(warriors.has("Oda"), "And Oda");                            //#C
assert(warriors.has("Tomoe"), "Tomoe, last but not least");        //#C

assert(warriors.size === 5, "There are 5 warriors in total");      //#D
```

#A Create an array of `ninjas` and `samurai`. Notice how Hattori is both a ninja and a samurai
#B Create a new set of warriors by de-structuring ninjas and samurai
#C All our ninjas and samurai are included in the new warriors set
#D There are no duplicates in the new set, Hattori even though he is contained in both the ninjas and the samurai sets is included only once!

In listing 3, we first create an array of `ninjas` and an array of `samurai`. Notice how `Hattori` is leading a very busy life; samurai by day, and a ninja by night. Now imagine that we need to create a collection of people that we can call to arms if a neighboring daimyo decides that his province is a bit cramped. We'll create a new set of `warriors` that will include all `ninjas` and all `samurai`. However, since `Hattori` is in both collections, we want to include him only once, it's not like two `Hattoris` will respond to our call.

In this case, sets are perfect! We don't need to manually keep track of whether an item has been already included, the set takes care of that by itself, automatically.

When creating this new set, we've used the spread operator: `[...ninjas, ...samurai]` (remember Chapter 4) to create a new array that will contain all `ninjas` and all `samurai`. In case you're wondering, `Hattori` will be present twice in this new array. However, when we finally pass that array to the `Set` constructor, `Hattori` will be included only once, see the following figure.

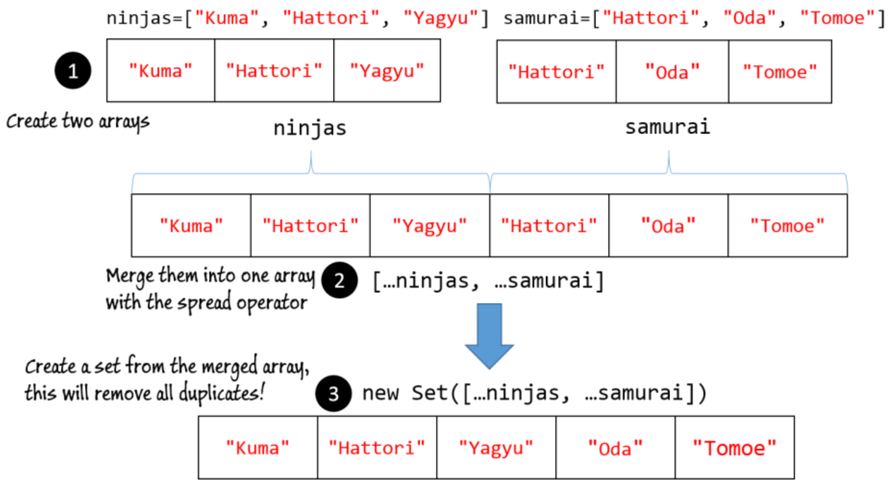

ninjas=["Kuma", "Hattori", "Yagyu"]
samurai=["Hattori", "Oda", "Tomoe"]
1
Create two arrays
ninjas
samurai
"Kuma" "Hattori" "Yagyu" "Hattori" "Oda" "Tomoe"
Merge them into one array with the spread operator
2
[…ninjas, …samurai]
Create a set from the merged array, this will remove all duplicates!
3
new Set([…ninjas, …samurai])
"Kuma" "Hattori" "Yagyu" "Oda" "Tomoe"

Figure 2 A union of two sets keeps the items from both collections (without duplicates, of course)

Intersection of sets

The second operation that we'll explore is the *intersection* of two sets `A` and `B`, that creates a set that contains elements of `A` that are also in `B`. In our example, this could be finding out `ninjas` that are also `samurai`, see the following example.

Listing 4 Intersection of sets

```
var ninjas = new Set(["Kuma", "Hattori", "Yagyu"]);
var samurai = new Set(["Hattori", "Oda", "Tomoe"]);

var ninjaSamurais = new Set(
  [...ninjas].filter(function(ninja){              //#A
    return samurai.has(ninja);                     //#A
  })                                               //#A
);

assert(ninjaSamurais.size == 1, "There's only one ninja samurai");
assert(ninjaSamurais.has("Hattori") == "Hattori is his name");
```

#A Use the spread operator to turn our set into an array, so that we can use the array's filter method to keep only ninjas that are contained in the samurai set

The idea behind listing 4 is to create a new set that will contain only `ninjas` that are also `samurai`. We'll do this by taking advantage of the array's `filter` method which, as you remember, creates a new array that contains only the items that match a certain criterion. In our case, that criterion is that the ninja is also a samurai (that it is contained within the set of samurai). Since the `filter` method can only be used on arrays, we have to turn our `ninjas` set into an array by using the spread operator:

```
[...ninjas]
```

Finally, in the end, we check that we've found only one ninja that's also a samurai, our jack-of-all-trades, Hattori.

Difference of sets

The final set operation that we're going to study is the difference of two sets `A` and `B`, that contains all elements that are in set `A`, but that are *not* in set `B`. As you might guess, this is very similar to the intersection of sets, with one small but significant difference. Let's take a look at the next listing, in which we want to find only true ninjas (and not the ones that also daylight? as samurai):

Listing 5 Difference of sets

```
var ninjas = new Set(["Kuma", "Hattori", "Yagyu"]);
var samurai = new Set(["Hattori", "Oda", "Tomoe"]);

var pureNinjas = new Set(
  [...ninjas].filter(function(ninja){              //#A
    return !samurai.has(ninja);                    //#A
  })                                               //#A
);

assert(pureNinjas.size == 2, "There’s only one ninja samurai");
assert(pureNinjas.has("Kuma"), "Kuma is a true ninja");
assert(pureNinjas.has("Yagyu"), "Yagyu is a true ninja");
```

#A With set difference, we care only about ninjas that are NOT samurai!

The only change that we have brought upon is specifying that we care only about the ninjas that are NOT also samurai, simply by putting an exclamation mark (`!`) before the `samurai.has(ninja)` expression.